XP 000610640

PD 11/1996
P. 44+46-49+51.

Endian-Neutral Software, Part 2

Program design and coding practices

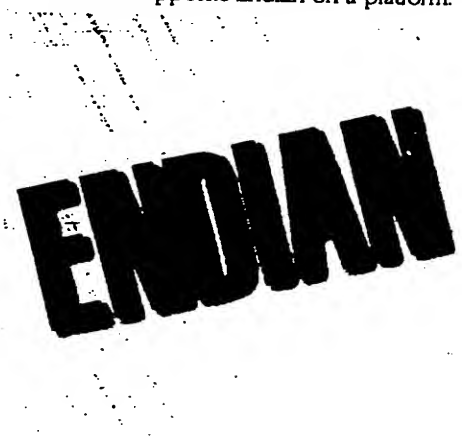
James R. Gillig

Software designed and written for Endian portability, commonly referred to as "Endian-neutral," should be recompile-and-run capable. Achieving Endian portability requires that you identify all necessary Endian dependencies in a program, separating and isolating them as best you can through interfaces and conventions. "Endian-aware" programs, on the other hand, may have well-defined, Endian-specific parts that require some modification for Endianness when porting the program to a processor of the opposite Endian; see Figure 1.

Endian-neutral (EN) design goals focus on separating Endian-neutral and Endian-specific parts, and minimizing the size of any necessary Endian-specific part and any necessary modifications for porting from a processor of one Endian type to another.

In general, Endian dependencies can occur when:

- An application provides Endian conversion for data interchange.
- An application has both a Big-endian (BE) and a Little-endian (LE) version of the same set of source code.
- An operating system manages a processor's Endian-related controls if any exist.
- Device drivers handle Endian differences between the processor the device driver runs on and its attached devices.
- Communications and LAN software handle Endian differences between connected systems.
- Conversion utilities allow the end user to import data from an opposite-Endian program.
- Compilers provide compiling options to generate different Endian-type code and data.
- Debuggers and dump utilities provide options for viewing data in different Endian forms.
- An instruction-set translator has to emulate the opposite Endian on a platform.



Jim is a software engineer for IBM in Boca Raton, Florida. He can be reached through the DDJ offices.

Endian-Neutral Programming

You can follow a number of practices for writing source code that's portable between BE and LE processors. Although the guidelines and examples I present here are based on C++, the principles apply to any language.

Use a high-level language. Programming-language keywords in a language such as C++ allow the declaration of data types and aggregate data and provide for correct data type conversion. Operations such as casting, union, and bit field allow flexibility and optimization for handling data but require Endian awareness to ensure the production of Endian-neutral code. High-level language constructs should be used for better programming consistency and handling of data.

Assembly language does not offer high-level constructs and data type checking, and is more difficult for architecture-neutral programming. However, using a high-level language is neither a necessary nor sufficient condition for making software Endian neutral and, in general, the EN guidelines covered here apply to assembly-language programming as well.

Object-oriented design and programming can provide a higher degree of neutrality than procedural programming alone because objects hide data from direct access by pointers from outside the object; however, the object class and its methods are still responsible for implementing Endian neutrality within the object's program code.

Use data types correctly. In general, the same data should not be used as different data types. Type conversion can corrupt data and introduce Endian dependencies. Different data types can have different byte lengths and the same data type may have different lengths on different processors.

A data type should be treated by executable program code as intended for that type. A multibyte-scalar integer, for instance, is treated by the processor as a single, indivisible data item that represents a numeric value. The location of bit and byte subfields within a scalar is variant between BE and LE, so treat data according to its data type, length, and Endian type. For portability, do not twiddle with the internal bits and bytes of multibyte-scalar data while dependent on their individual byte addresses.

Organize aggregate and scalar data. Program code becomes Endian dependent when a multibyte-scalar data item is treated as aggregate data containing multiple pieces of data across its bytes. Multiple pieces of data should be organized as aggregate data using proper programming-language constructs such as a data structure or data array. The individual members

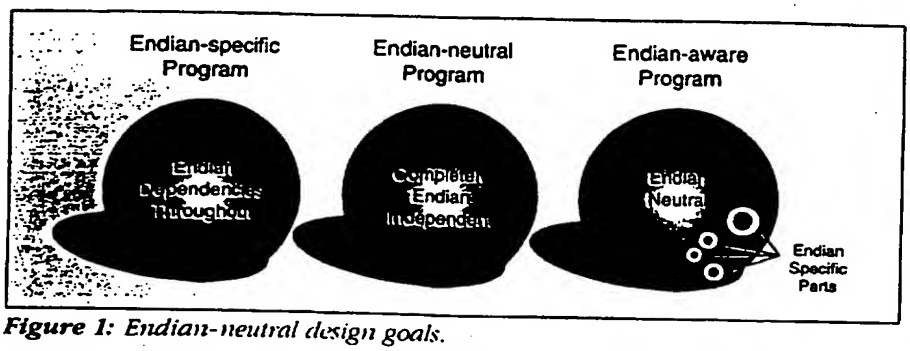


***Figure 1:** Endian-neutral design goals.*

(continued from page 44)
of a structure may be a scalar element or another aggregate data element with its own members.

You should be cognizant of aggregate data and scalar data when programming and treat scalars as single, indivisible, nonaggregate data items. Organize scalars as members of an aggregate data construct when dealing with multiple pieces of data in a collective manner.

Avoid pointers within scalars. Do not point or index to smaller units of storage within the internal byte structure of a scalar. LE data is byte-reversed BE data, so internal byte positions are different. Do not address, point, or index to a byte, halfword, or word scalar that may exist within a longer data type. For example, a short integer contains two bytes, an integer contains two short integers, and a long integer contains two integers.

Accessing bytes internal to scalar data through pointers and indexing is a poor practice; see Example 1. Multiple-byte data elements can be defined as a byte array such as *char p[4]*, which is independent of Endian.

Avoid overlaying scalars. A data item is said to be overlaid when it has more than one meaning or type. Examples of overlaid data include using: a short integer within a long integer; the high-order byte of an integer as a flag; a bit field and a binary value in the same integer; and a scalar as an array of bytes.

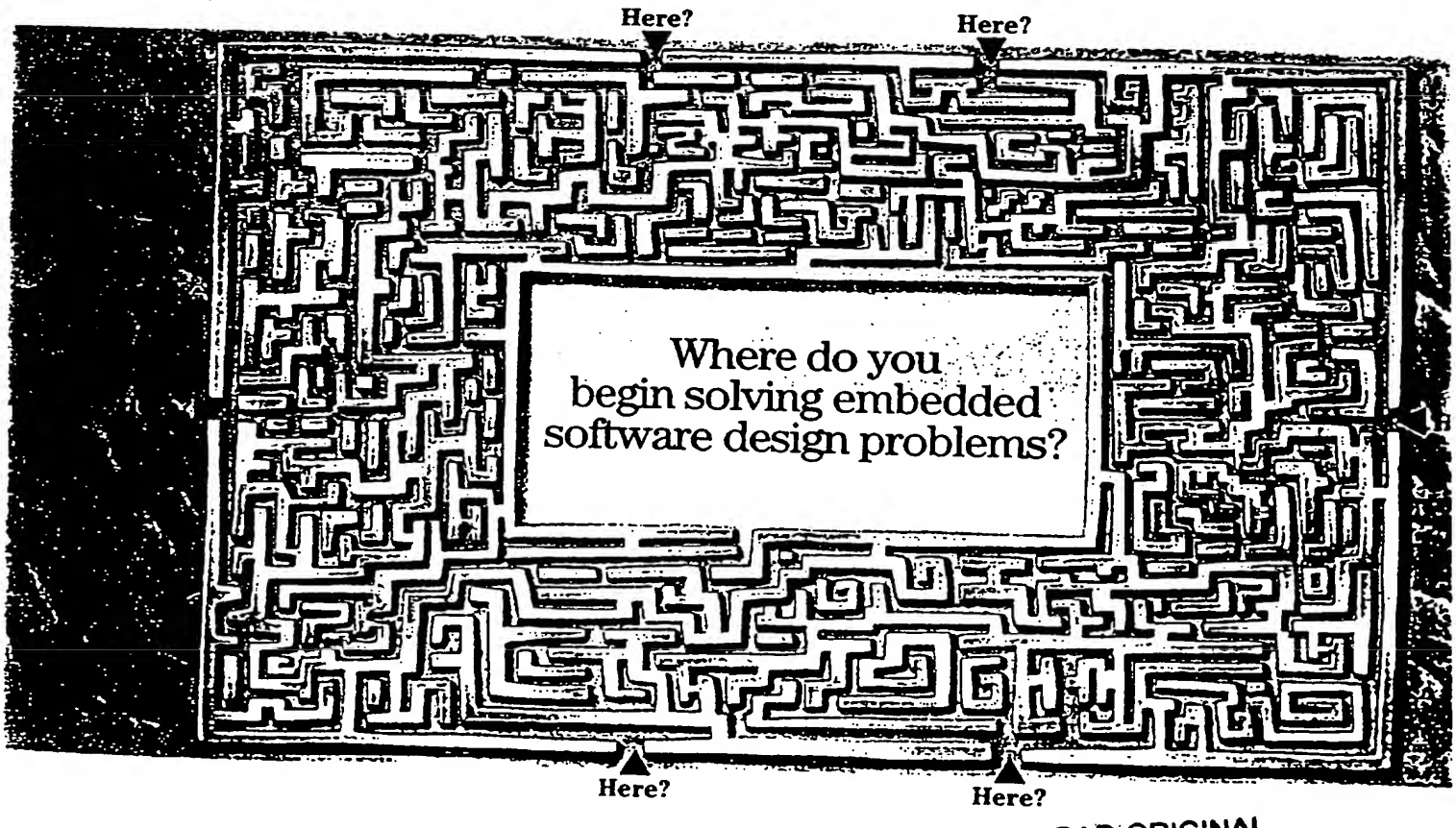
A longer data type should not overlay a shorter data type so that integers don't overlay characters or bit fields and longer integers don't overlay shorter integers. Overlaying data is not the same as sharing memory for mutually exclusive use by different data types. Sometimes programmers overlay data to save storage, but other times storage saving is not that valuable and extra program code may be necessary to get at the desired piece of data.

Be aware of Endianness with casting, union, and bit fields. C++ provides programming constructs that may have side effects on data in the same or different Endian modes.

Casting forces a data-type conversion to another type: for example, *short* to *long* or *char* to *short*. Pointer or reference casts allow a program to view a variable of one data type as if it were another type. Thus, they make program code dependent on the Endian mode of execution, as in Example 2(a), where the cast to a short

```
long  a = 0x01020304;
char  c, *p;
p = (char *) &a; //  set up pointer to a
c = p[2];        //  c is 0x03 in BE mode
                 //    and 0x02 in LE mode
```

***Example 1:** Avoiding pointing and indexing within scalars.*



pointer results in pointing to different data depending upon whether the execution mode is BE or LE. Regardless of Endian, do not cast a longer type to a shorter type because if the value does not fit in the smaller type, it can lead to data corruption as a result of type conversion; see Example 2(b).

Union allows different variables and data types to share the same memory. Don't use different data types concurrently in union, and do not use union to do type conversion. The actual data currently stored in union, its actual data type, and its referenced data type must be consistent. In Example 3, for instance, the array name *p* should not be used to access data stored with variable *a* because they are different types (*p[0]* will access the MSB of integer *a* in BE mode and the LSB of *a* in LE mode.

A bit field is a set of adjacent bits that allows efficient, convenient use of memory for implementing a program's flags, switches, or discretes. Example 4(a) shows a bit field defined in an integer. Fields can be referenced by name, such as *bitfield.b*.

If bit-field data is created with the structure in Example 4(a) and ported to an opposite Endian system, then either the order of the actual data bit fields *a, b, c, pad* in the integer must be reversed to *pad, c, b, a* for use with the same structure or the order of the bit fields defined within the structure must be reversed for use with the original imported integer and bit-field ordering; see Example 4(b).

The reversal effect upon bit-field ordering between BE and LE is analogous to byte reversal between BE and LE; in general practice, it's simply extended down to the bit-field level by the "programming language." However, this practice cannot be guaranteed for all compilers.

Alternatively, a program can perform its own bitwise-logical operations to select, set, and test bit fields by shifting and masking. It must not depend on byte address and order, however. That would make it Endian specific and not portable to an opposite-Endian platform.

Avoid alignment complications. Data alignment is a general portability concern and, although independent of Endian, it

```
(a) long a;
    short b;
    a=1;
    b=*(short *)&a;
                // b=0 for BE and b=1 for LE

(b) int a;
    char b;
    b=(char) a;
    // Data loss. but no Endian problem
```

Example 2: (a) Making program code depend on Endian execution mode; (b) data corruption due to type conversion.

```
char c;
union  udata  {
  int a;
char p[4];
}
uvar :
uvar.a = 0x01020304;
c = uvar.p[1] ;  // c is 0x02 in BE mode
                 and 0x03 in LE mode
```

Example 3: The array name p *should not be used to access data stored with variable* a *because they are different types.*

```
(a)
struct {
   unsigned a   : 2 ;  // two bit field
   unsigned b   : 1 ;  // one bit field
   unsigned c   : 3 ;  // three bit field
   unsigned pad : 10 ; // ten bit padding
} bitfield;

(b)
struct {
   unsigned pad : 10 ; // ten bit padding
   unsigned c   : 3 ;  // three bit field
   unsigned b   : 1 ;  // one  bit field
   unsigned a   : 2 ;  // two bit field
} bitfield;
```

Example 4: (a) Bit fields defined in an integer; (b) reversing the order of data bit fields.

can complicate assumptions about byte location. A data item is "aligned" when its address is a multiple of its byte length. Thus a 2-byte short integer aligns on an address that is a multiple of 2; a 4-byte long integer aligns on an address that is a multiple of 4; and an 8-byte long integer aligns on an address that is a multiple of 8. If the architecture requires it, compilers will align data by default. When unaligned, a data item may be at a different location than when aligned.

Assumptions about alignment, byte location, and Endian byte order can become more complex, as in Example 5. When aligned, a padding byte may be inserted by the compiler after the character "c" to force the integer *n* onto a proper even-address boundary.

In Example 5, the pointer *p* may end up pointing to a pad byte before *n* if the data is aligned, to the most significant byte of *n* if the data is unaligned and BE, or to the least significant byte of *n* if the data is unaligned and LE.

To reduce padding for aligned data, avoid intermingling different data-type lengths by arranging data in order from longer to shorter types.

The Endian Test

An "Endian-adaptive" program queries or tests the processor to decide on the Endian mode of its current execution so as to take a LE processing path or BE processing path at run time. If the processor does not provide a means for telling software its Endian mode, then a program test can be improvised, as in Example 6(a), which implements an "Endian test." Endian macros can be defined for reuse in a program, as in Example 6(b).

```
struct {
    char c;
    // alignment pad byte may occur
    // between c and n
    integer n;
} *p;
p=(char*)&a.c;
// p points to what byte?
```

Example 5: Dependency on alignment and Endian mode of execution.

```
(a) #define BE 1
    #define LE 0
    char endian (void)
    {
        short x = 0x0100;
        return *((char *) &x);   // return 0x01 for BE and 0x00 for LE
    }

(b) #define is_big_endian    (endian() == BE)
    #define is_little_endian (endian() == LE)
```

Example 6: (a) Endian test program; (b) Endian macros defined for reuse.

```
(a) long a=0x01020304;
    char c,*p;
    p=(char *) &a;  // set up pointer to a
    c=p[2];         // c is 0x03 in BE mode and 0x02 in LE mode

(b) if(is_big_endian) c=p[1]; else c=p[2]; // c is 0x02 in BE or LE.

(c) c=(a & 0x00FF0000) >> 16; // reliably yields 0x02 without a test
```

Example 7: Making code neutral with the Endian test.

Endian-adaptive code is Endian neutral in that the same source can be recompiled without change to run in the other Endian mode. Recompilation is necessary for a different processor instruction set as well as Endian type.

As an example of how to make code neutral with the "Endian test," consider Example 7(a). This can be written using the macro, as in Example 7(b), or without the macro and independent of byte order (for this particular example), as in Example 7(c). This example assumes *long* is four bytes, but this is not a safe assumption for portability. An alternate implementation of this example is to declare *long a* instead as a character array *a[4]*, in which case *a[2]* is the same data 0x03 in either Endian mode, as single-byte character data has no Endianness.

You must be careful with an adaptive implementation to make sure it is portable for BE and LE across the required processors and compilers. You may be able to implement necessary Endian-specific code with more-portable Endian-adaptive code; nevertheless, writing Endian-neutral code is the first step in achieving portable software across platforms of different Endian types.

Application Portability

There are different ways to port a program to run on other processors. The most common is to recompile the source code to run on the new processor; another way is to translate the original binary code and its nonnative processor instruction set on the new processor.

A program can be recompiled to run on the instruction set of a different processor (recompile-and-run). If the new processor is of the opposite Endian, then the source code (unless it's Endian neutral) will need to have its Endian-specific code modified to run in the new Endian mode before recompilation. Recompiling an application to another processor produces a new binary version of the program that will have to be serviced and supported. A given application may have different binary versions for running on different processors and in different Endian modes. The application's source-code set, any Endian-specific parts, and all binary versions of it will have to be maintained.

Translation of binary code requires an instruction-set translator (IST) to translate the original processor's instruction set and Endian mode on the new (nonnative) processor. Compiled, binary code undergoing translation is treated as data by the IST. If the new processor is of the opposite Endian then the IST, running in the Endian mode of the new processor, must handle the byte-reversed instructions and data of the program being translated.

A special consideration is a bi-endian processor. A single binary version of an Endian-neutral program cannot run in both the BE and LE modes of a bi-endian processor even though the instruction set is the same. This is because the address model, and therefore the (scalar) byte order, is different for both executable instructions (binary code), and data. The benefit of a bi-endian processor is that existing LE and BE applications and their operating systems can be ported to it and still run in their native Endian mode. An application must be compiled to the Endian of the operating system and processor on which it will run.

Today's modular systems share and reuse software resources, so be aware of the Endian effect on reusable resources and make them Endian neutral wherever possible. Presentation resources (icons, dialogs, controls, and image bitmaps), program resources (pointers, integers, arrays, structures, and headers), and miscellaneous resources (device drivers, presentation drivers, objects, server resources, and files) are created and passed around for reuse in different programs and on different platforms. These resources can introduce Endian dependencies into a program, so awareness, inspections, and testing are imperative.

Data P rtability

Users of today's open, connected systems need the capability to interchange data between different systems. Data can be interchanged through media (diskettes, disk, tape, and so on) and local- and wide-area networks of client, server, and host communications. Data portability requires the ability to handle any difference between the Endian type of the data being

ported and the Endian mode of the system to which it's being ported.

Data conversion between BE and LE requires knowing the data type, length, and Endian type for all data elements of an aggregate data layout to be converted (such as a data structure, array, file, or table). The conversion algorithm between BE and LE is straightforward: Reverse the byte order of a given multibyte-scalar data item. A prerequisite for conversion and often the crux of the conversion problem is knowing the aggregate data layout or organization of the data to be converted. Data is often understood only by the application that creates it.

When importing data created or processed by a program running on another platform, the data's Endian type and other data characteristics may need to be converted. Typically, the receiver or importer of the data does the conversion; this is known as "receiver-makes-right."

Data can be readily interchanged between the same application running on different platforms because an application understands its own data. So a BE version and LE version of the same application running on two different platforms can exchange and convert data to the correct Endian type of the application's resident platform.

To interchange data between different applications, a conversion utility can be written that understands an application's data and converts it to another application's data format and Endian type. Datafile formats often become public for popular applications. For example, different versions of a word processor that runs on an IBM PS/2 (LE, Intel), Macintosh (BE, Motorola), and a RISC platform (bi-endian PowerPC) may need to import data from another word processor.

Endian-neutral design goals focus on separating Endian-neutral and Endian-specific parts

A self-describing data resource can be created, handled, and converted by any program that understands the public specification of its format and data descriptors. For example, if a piece of data has associated descriptors for its data type, length, and Endian type, then it can be easily converted.

Data can be kept in a standard ("canonical") form for easier conversion. For example, if a program always writes its data as LE, then a program running on a BE platform knows to convert that data from LE to BE upon reading the data from some magnetic/optical media, network, or communications link.

Text Data and Unicode

A text stream or binary stream is composed of byte data. A byte is the smallest addressable unit of storage; therefore, character data has no Endianness and is neutral and portable. This is for single-byte character encodings handled by the character data types (signed and unsigned).

The Unicode standard defines a fixed-width, uniform-text, character-encoding scheme. All Unicode characters are represented as 16-bit values. For example, the hex value 0x0041 represents letter A; 0x0020, the space character and 0x0409, the character named "CYRILLIC CAPITAL LETTER LJE." Unicode values must be treated as single, 16-bit, unsigned integer values. Like other multibyte-scalar values, Unicode data stored as binary data in a

(continued from page 49)

file, or other interchange media, is Endian dependent. To manage Unicode data correctly, data-interchange programs must be sensitive to the Endianness of the source and target platforms.

To help deal with Endian conversion, the Unicode standard describes an optional technique for programs that manipulate Unicode data. The standard defines the value 0xFEFF as the character named "BYTE ORDER MARK" (BOM). Applications may use this character to explicitly announce the Endianness of the data. The byte-swapped mirror image of the BOM, the value 0xFFFE, is not a defined Unicode character. Therefore, an application expecting the BOM and finding 0xFFFE instead, knows that the Unicode data is not in the Endian format expected. The application may then decide to perform byte swapping to convert the Endian type or notify the user to run a conversion utility.

Bit-Field Data

Data portability and compiler implementation of bit fields is another issue. Within an integer or word, fields may be assigned in left-to-right order by some compilers and right-to-left by others. The programmer must contend with these complications in addition to Endian byte order when exchanging data across systems. In general, using a 32-bit word, BE labels bits 0–31 beginning with the most significant or leftmost bit; LE labels bits 0–31 beginning with the least significant or rightmost bit.

Endian-adaptive code is Endian neutral in that the same source can be recompiled, without change, to run in the other Endian mode

A bit field is a contiguous set of bits, where the most significant bit is on the left end and the least significant bit is on the right end. The programming language, if it supports bit fields, will generally extend the Endian type for compiled data down to the bit-field level, even if a processor doesn't support bit fields; that is, multiple bit fields defined within a word appear in left-to-right order for BE and right-to-left order for LE. This is the prevailing practice for compilers.

So three different bit fields named *a,b,c*, appearing in that order and beginning at the left end of a word for BE would appear in *c,b,a* order, beginning at the right end of a word for LE. This is for bit *fields*, not the bits themselves; they retain the same relative bit order within a defined field whether BE or LE. For example, if 16 1-bit fields are defined within a 16-bit word for LE, then all the bits in that word will in effect be in reverse order when compared to its data representation for BE.

When importing bit-field data from a system of the opposite Endian type, either the order of the data bit fields must be reversed or the bit fields must be accessed by a program in the reverse order.

Acknowledgments

Thanks to the following people at IBM for their contribution and review: Art Adkins, Ken Borgendale, Norman Cohen, Ian Holland, Alan MacKay, Roy Ritthaler, Rick Simpson, and Mark Wieland.

DDJ

To vote for your favorite article, circle inquiry no.